MA 3046 - Matrix Analysis
Floating-Point Numbers and Machine Accuracy

The Floating Point Representation

It is a well-known fact that every real number can be represented in decimal form, although often doing so may require an infinite number of digits. For example:

$$\begin{aligned} 21\tfrac{7}{16} &= 21.4375 \\ 7\tfrac{1}{3} &= 7.3333\ldots \quad . \end{aligned}$$

Alternatively, we can use (decimal) scientific notation to uniquely represent every real number in the form:

$$x = \pm\,.\,a_1 a_2 \ldots a_n a_{n+1} \ldots \times 10^s$$

where, unless $x = 0$,

$$1 \le a_1 \le 9$$

and

$$0 \le a_i \le 9\ ,\ i = 2, 3, \ldots$$

For example:

$$21\tfrac{7}{16} = .214375 \times 10^2 \quad .$$

Moreover, an equivalent number system can be constructed based on, instead of 10, *any* positive integer ≥ 2. In other words, every real number can be expressed:

$$x = \pm\,.\,d_1 d_2 \ldots d_n d_{n+1} \ldots \times \beta^s \quad ,$$

where β (≥ 2) is called the *base*, the digits $d_1 d_2 \ldots d_n d_{n+1} \ldots$ are called the *mantissa* and s is called the *exponent*. In this case the digits of the mantissa satisfy

$$1 \le d_1 \le \beta - 1$$

(again unless $x = 0$) and

$$0 \le d_i \le \beta - 1\ ,\ i = 2, 3, \ldots \quad ,$$

Thus, for example, in base 2,

$$\begin{aligned} 21\tfrac{7}{16} &= 16 + 4 + 1 + \tfrac{1}{4} + \tfrac{1}{8} + \tfrac{1}{16} \\ &= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &\qquad + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} \\ &= +10101\,.\,0111\ _b \\ &= (+\,.\,101010111 \times 2^5)_b \quad , \end{aligned}$$

where $(\ )_b$ denotes a binary number. (Note that binary representation has one unique feature. Unless $x = 0$, the leading digit (d_1) is always equal to 1. We shall see later that this has some interesting consequences!)

The key difference between the theoretical representation of real numbers in scientific form and the reality of electronic computers is that a computer must utilize a unique location in its memory to store each digit of a number, and therefore real computers cannot store the infinite number of digits necessary to exactly represent most real numbers. Thus, real computers generally store, instead of the "true "real number, the "closest" real number that can be represented by some *fixed number of digits*. (On any given computer, the base (β) and number of digits used to represent real numbers are generally predetermined by a combination of hardware and software (e.g. FORTRAN) design choices.) We call this form the **normalized floating point representation** of x , and denote it as:

$$fl(x) = \pm \, . \, d_1 d_2 \ldots \tilde{d}_n \times \beta^s \quad .$$

In this representation, d_1 is referred to as the *most significant digit*, and $\tilde{d}_n$ as the *least significant digit*. It is rather easy to show that floating point representation allows precisely $[2(\beta - 1)\beta^{n-1} + 1]$ **different** values for the significant digits (mantissa). Therefore, in general:

$$x \neq fl(x) \; .$$

In order to form $fl(x)$, i.e. to determine the "closest" floating point number to a given real number x, most computers use one of two methods:

(1) Let $\tilde{d}_n = d_n$ and simply ignore the digits $d_{n+1} d_{n+2} \ldots$. This is called *chopping*.

(2) Let $\tilde{d}_n = d_n + 1$ if $d_{n+1} \geq \frac{\beta}{2}$, otherwise chop. This is called *rounding*.

Thus, for example, in a four-digit computer using base 10,

$$21\tfrac{7}{16} = \begin{cases} .2143 \times 10^2 \text{ (chopped)} \; , \\ .2144 \times 10^2 \text{ (rounded)} \; . \end{cases}$$

For most of the remainder of our examples, we shall use $\beta = 10$ because of its familiarity and ease of hand computation. However, most computers actually use bases other than 10, the most common being 2 (*binary*), 8 (*octal*) and 16 (*hexadecimal*). (Single precision FORTRAN on IBM mainframes uses $\beta = 16$ and $n = 6$. By contrast, most UNIX workstations use $\beta = 2$ and $n = 33$ for single precision.)

We would close this section by noting three other seemingly minor, but really rather important points, that many textbooks do not cleanly address. The first is that, in floating point representation, *leading zeros are irrelevant, but trailing zeros are important*. For example, the numbers $.2143 \times 10^2$ and $.002143 \times 10^4$ are equivalent (and virtually every existing computer would immediately convert the second to the first), but $.2143 \times 10^2$ and $.214300 \times 10^2$ are **not**, since the first implies (in a chopping machine) a floating point number for which

$$21.43 \leq fl(x) < 21.44$$

while the second implies that the floating point number satisfies

$$21.43 \leq fl(x) < 21.4301$$

The second important point is that *there is no a priori relationship between the number of digits in a normalized floating point number and the number of digits after the decimal*

point when that number is written in standard form. Specifically, for example, 0.3333 and 7.3333 are both accurate to four decimal places, but the second implies the use of a five-digit accurate floating point machine, while the first only implies a four-digit accurate one.

Lastly, *a number which requires only a finite number of non-zero digits to exactly represent in one base may require an infinite number in another.* This can easily lead to somewhat unexpected results. For example, in either base 2 or 16, the real number $x = \frac{1}{10}$ is not expressible exactly as a floating point number. Thus, in a hexadecimal computer, a **DO** loop like:

```
DELTA = 0.1
DO 10 I = 1,1000
    X = X + DELTA
      ...
```

will not end with the value $X = 100$!!!!

Measuring Errors

A principal aspect of numerical analysis is the *quantitative* consideration of the errors which arise due to floating point representation. If fact, for reasons we shall discuss shortly, error analysis actually involves the consideration of several different measures of error. We shall illustrate the most commonly used of these by considering the error involved in simply approximating a given real number by the closest floating point number. The *(actual) error* (which we generally call just the error) is:

$$x - fl(x) \quad .$$

In may cases, however, this value is not especially illuminating. For example, the same actual error arises with either:

$$fl(1) = 1001 = +.1001 \times 10^4 \ ,$$

or

$$fl(999,000) = 1,000,000 = .1 \times 10^7 \ .$$

Clearly, in many contexts (e.g. financial), most reasonable people would not accept these as being equally accurate. Thus, it is often more relevant to speak of the *relative error*

$$\frac{x - fl(x)}{x} \ , \quad x \neq 0 \ .$$

(Note that neither of these two measures is it inherently better - they are simply two parts of the same puzzle. Relative error, for example, frequently provides only limitedly useful information when x is close to zero.)

Unfortunately, with the exception of highly artificial textbook or classroom examples, there exist few, if any "real" problems where either the actual error or the relative error

are known, or even computable. In fact often even their algebraic signs are unknown! The best that one can generally do is to estimate their magnitude. Therefore, error analysis also frequently involves the absolute values of the two quantities above, i.e. the *absolute error*:

$$|x - fl(x)|$$

and *absolute relative error*:

$$\left| \frac{x - fl(x)}{x} \right| .$$

(Note that these last two measures correspond precisely those traditionally used to report experimental results, e.g. when we write either

$$x = .3152 \pm .0005 ,$$

or

$$x = .3152 \pm .16\% .)$$

Moreover, note that the definition of relative error implies the alternate, but equivalent, statement about floating point representation:

$$fl(x) = x(1 + \delta) .$$

Lastly, it is fairly easily shown that the relative error in converting x to $fl(x)$ satisfies:

$$|\delta| \leq \begin{cases} \beta^{1-n} & \text{(chopping architecture)} \\ \frac{1}{2}\beta^{1-n} & \text{(rounding architecture)} \end{cases} .$$

In any given computer, the quantity on the right hand side of this expression is commonly referred to as the *machine precision* of that computer, and, as is easily seen, is a function of both the base and the number of significant digits retained. Machine precision is a fundamental concept in numerical analysis. It expresses, in one sense, the "best case" behavior of a given machine, i.e. while a particular computation *may* produce an answer that is more accurate than machine precision, you never have a right to *expect* that.

Errors and Arithmetic Operations

Of course, a computer which did nothing but read in a real number x, convert it to floating point form, $fl(x)$, and output the result would be of little practical interest. The main strength of the digital computer is its ability to perform (many) arithmetic operations very rapidly. Thus, the more interesting and relevant issue is what impact does the floating point representation of real numbers have on familiar arithmetic operations? As we now show, the answer depends greatly on which operation we are talking about.

The consequences of using floating point numbers for multiplication are the easiest to analyze. Suppose, for example, a hypothetical computer were to try to calculate (xy), using rounding arithmetic. The computer would first have to form the two numbers:

$$\begin{aligned} fl(x) &= x(1+\delta_1) \ , \quad |\delta_1| \leq \tfrac{1}{2}\beta^{1-n} \\ fl(y) &= y(1+\delta_2) \ , \quad |\delta_2| \leq \tfrac{1}{2}\beta^{1-n} \quad , \end{aligned}$$

each of which would be in theory n digits long. Next, the computer would form the product

$$fl(x) \cdot fl(y) \ ,$$

which could (theoretically) be up to a $2n$ digit number. (Many computers have so-called *double length accumulators*, which allow this product to be formed without error.) But then, to store the result, the computer would first have convert it to n digits. This conversion would introduce another error, on the order of machine precision. We can summarize the above discussion in terms of the following formula, which expresses what is actually stored as the computed value for xy,

$$fl(fl(x) \cdot fl(y)) = fl(x)fl(y)(1+\delta_3) \ , \ |\delta_3| \leq \tfrac{1}{2}\beta^{1-n} \quad .$$

Thus, instead of xy, the computer actually has stored:

$$\begin{aligned} & fl(x)fl(y)(1+\delta_3) \\ & \qquad xy(1+\delta_1)(1+\delta_2)(1+\delta_3) \\ & \qquad\quad \doteq xy(1+\delta_1+\delta_2+\delta_3) \\ & \qquad\quad = xy(1+\delta_m) \ , \quad |\delta_m| \leq \tfrac{3}{2}\beta^{1-n} \quad . \end{aligned}$$

But this implies that δ_m is of effectively the *same order of magnitude* as machine precision. Therefore we claim that multiplication does not generally significantly distort the relative error. A basically similar result can be shown for division.

However, the situation with addition and subtraction is not nearly as clean. The difficulty with these operations is that before two numbers can be added or subtracted, they must first be rearranged so that both *have the same exponent.* (This is why the old slide rule was only useful for multiplication and division.) In a computer, this alignment can usually only be accomplished by shifting the significant digits of the smaller (in magnitude) of the two numbers to the right, inserting leading zeros and chopping or rounding the trailing digits so the overall length of the number remains unchanged. For example,

$$\begin{array}{rcrcr} & .2157 \times 10^3 & \rightarrow & & .2157 \times 10^3 \\ + & \underline{.3024 \times 10^2} & \rightarrow & + & \underline{.0302 \times 10^3} \\ & & & & .2459 \times 10^3 \end{array}$$

While a detailed analysis similar to that for multiplication can also be conducted here, it is really not necessary, because it should be obvious that *for addition, as long as both numbers have the same algebraic sign, the error introduced by the right-shifting occurs only*

in the least significant digit of the result, and therefore cannot significantly exceed machine precision. Therefore, we can conclude that addition of two floating-point numbers with the same algebraic sign is numerically "safe."

Unfortunately, the same conclusion cannot necessarily be drawn for subtraction (or for the addition of two numbers with differing algebraic signs), as the following example, again in a four-digit chopping decimal machine, illustrates:

$$\begin{array}{rcl} 21.374 & \rightarrow & .2137 \times 10^2 \\ - \ \underline{21.366} & \rightarrow & - \ \underline{.2136 \times 10^2} \\ .008 = .8000 \times 10^{-2} & & .0001 \times 10^2 = .1000 \times 10^{-1} \end{array}$$

The relative error here is significantly above machine precision. The "problem" of course should be fairly obvious - because of the cancellation, *the least significant digits of the original numbers have in fact become the most significant digits of the result.* In a signal processing sense, if we think of the most significant digits of a number as representing the "signal," and the least significant digits as somewhat analogous to the numerical "noise," what has occurred in this calculation is that the "signal" has been almost totally canceled out, and the resulting value is almost pure "noise." (Obviously, not **every** subtraction will result in a situation this extreme. But such behavior is **possible at any time.**) This phenomenon, when it occurs, is commonly referred to as *catastrophic cancellation.* Catastrophic cancellation represents perhaps the single most significant potential drawback in numerical computation.

Unfortunately for computer calculation, there is generally no way to recover from catastrophic cancellation once it has occurred. Therefore, most practical numerical strategies involve a combination of:

(1) Prevention, i.e. the use, as much as possible, of "good" numerical procedures which will, ahead of time, minimize the possibility of a catastrophic cancellation calculation occurring, and
(2) Identification, i.e. the use of numerical procedures which will at least after the fact warn the user that a result may have been significantly degraded by cancellation errors, and therefore is not to be trusted.

Preventative measures generally involving reformulating calculations, by using expressions which, while *mathematically equivalent*, are **not** numerically equivalent. For example, because we have shown division is numerically "safer" than subtraction, it is almost always preferable that "small" numbers be computed not by the subtraction of two nearly equal quantities, but instead by the division of a "small" quantity by a "large" one. Therefore, while the expressions:

$$\sqrt{x+1} - \sqrt{x}$$

and

$$\frac{1}{\sqrt{x+1} + \sqrt{x}}$$

are algebraically equivalent, the second is clearly numerically superior when x is large relative to unity.

Unfortunately, strategies for identifying when catastrophic cancellation has occurred tend to be more method-specific, and therefore will be discussed in more detail in the context of appropriate methods. Furthermore, even when such strategies exist, they usually only indicate, *after the fact*, that catastrophic cancellation has occurred. But by that time, whatever numbers remain are almost total "noise," and it is almost impossible to recover any significant digits. The only choice is to go back, try and reformulate the problem into a more numerically stable form, and rerun it.

Another important aspect of floating point computer arithmetic is that most of the familiar "rules" of arithmetic generally *fail to hold* in the exact sense. For example, most of the time

$$[fl(x) + fl(y)] + fl(z) \neq fl(x) + [fl(y) + fl(z)] \ , \quad \text{etc.}$$

Even more intriguing is the fact that, due to differences in the way computer software is written, the **same expression**, evaluated on two different computers may yield **different results**, even when both use the same floating point representation. (Although, if both expressions used numerically "safe" calculations, we would not expect these results to differ by significantly more than machine precision!)

One final area where errors due to floating point representation may have significant consequences is in the numerical evaluation of functions. Specifically, there are almost always two basic sources of floating-point errors when functions are evaluated:

(1) The functions must, in general, be *calculated*, e.g. when e^{-x} is approximated by the Taylor Series:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$$

and this calculation process may introduce exactly the kind of arithmetic errors described above. (We would note that, for reasons beyond our discussion here, the Taylor series turns out not to be a smart way to do this particular calculation.)

(2) Even if the arithmetic operations were able to be performed without error, we would still, in general, be computing not $F(x)$, but in reality $F(fl(x))$.

The second of these two effects may be further investigated by considering the absolute value of the relative error that arises due solely due to replacing x by $fl(x)$:

$$\begin{aligned} \left| \frac{F(x) - F(fl(x))}{F(x)} \right| &\equiv \frac{|F(x) - F(x(1+\delta))|}{F(x)} \\ &= \left| \frac{F(x) - F(x+\delta x)}{\delta x} \frac{\delta x}{F(x)} \right| \\ &= \left| F'(\zeta) \frac{x}{F(x)} \right| |\delta| \quad . \end{aligned}$$

(The last equation results from the application of the mean value theorem for the derivative, and ζ is some point between x and $x + \delta x$.) But a close inspection of this last formula clearly indicates that the resulting relative error is equal to machine precision *multiplied*

by another factor. (In engineering terminology, this multiplying factor might be referred to as the *gain* of the procedure.) Moreover, this factor may be large, and therefore the computed function value may be significantly in error (relative to machine precision), if any one (or more) of the following are true:

(1) x is "large,"
(2) $F'(x)$ is "large," or
(3) $F(x)$ is "small."

(Actually, function evaluations usually involve one further important source of errors, although one not related to floating point representation. These errors arise because many functions, e.g. $\sin(x)$ and e^x are *transcendental*, i.e. their values can in general only be exactly calculated by the Taylor Series, e.g.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Therefore, in practice such infinite series are replaced by appropriate approximations, e.g. *truncated partial sums*. We shall have further occasion to study this latter type of errors in this course.)

The IEEE Floating Point Representation

As noted above, the IEEE floating point representation of the number x , which is used in most UNIX workstations is:

$$fl(x) = \pm\, d_1 \,.\, d_2 d_3 \ldots d_n \times \beta^s \quad ,$$

where β is two (binary), and, unless $x = 0$, the leading digit (d_1) equals one. Therefore, in these machine, only the digits $d_2, d_3 \ \ldots \ , d_n$ are actually stored. Representation of a single precision real floating point number requires 32 bits, while 64 are used for double precision real numbers. The leading bit always represents the sign of the number ("0" if the number is positive, and "1" if it's negative). The next eight bits (eleven for double precision) store the so-called *biased* exponent, s+127 (s+1023 for double precision). The remaining digits (twenty-three for single precision and fifty-two for double) are for the mantissa or fraction of the floating point number. (Because the leading significant digit, d_1, is not stored, the *effective* accuracy of IEEE numbers is actually one digit more than the number stored, i.e. twenty-four binary digits for single precision and fifty-three for double.)

Thus, for IEEE single precision, the digits satisfy:

$$d_1 = 1$$

$$0 \le d_i \le 1, \quad 2 \le i \le 24 \quad ,$$

$$-128 \le s \le 127 \quad .$$

(We generally express the resulting representation, however, as eight hexadecimal ($\beta = 16$) digits, rather than thirty-two binary ones. Note that in hexadecimal, 10 = a, 11 = b , etc.)

The actual conversion from decimal to binary is fairly tedious, since it involves repeated division and subtraction. For example, to convert:

$$\pi = 3.141592654\ldots$$

to hexadecimal, we start the largest power of 2 that is *smaller* than the given number (in this case 2). (This serves to define the exponent.) Then, we find the corresponding digit by dividing the full number by two raised to that power, and discarding the remainder, i.e.

$$\pi/2 = 1.57 \quad \Rightarrow \qquad d_1 = 1 \quad \text{and} \quad s = 1 \ ,$$

(This leading digit, of course, will not be stored in the actual representation of the number.) Subtracting 2^1 from π leaves the remainder:

$$1.141592654\ldots$$

which is then divided by the next lower power of 2, i.e. 2^0 , and the remainder again discarded. The resulting integer part (if any) becomes d_2, the first binary digit actually stored. Then, subtracting this digit from the original remainder leaves us with the new remainder

$$0.141592654 \ .$$

This new remainder is now divided by 2^{-1}, then chopped to yield the next digit, i.e.:

$$.141592564 \div 2^{-1} = .2831853\ldots \quad \Rightarrow \qquad d_3 = 0 \qquad ,$$

and the process continues. The final result is:

$$\pi \ = \ 3.141592654\ldots \ = \ (\ +1.10010010000111111011011\ldots \times 2^1)_2 \qquad .$$

(As noted before, the twenty-three bits to the right of the decimal are the only portion of the mantissa we'll actually store in the computer representation of π.)

To finish determining how π will be stored in single precision, we note that the exponent in this case satisfies

$$s + 127 = 1 + 127 = 128 = (10000000)_2$$

and the leading (sign) bit will be a zero since π is positive. Therefore, starting with (0), appending first the eight bits for the biased exponent $(10000000)_2$, and then the remaining twenty-three bits of the mantissa, we see that we can write the IEEE single-precision machine representation of π as

$$\underbrace{0}_{\text{sign}} \ \underbrace{10000000}_{\text{exponent}} \ \underbrace{10010010000111111011011}_{\text{mantissa}} \qquad .$$

Partitioning this into four bit hexadecimal digits yields

$$\underbrace{0100}_{4}\ \underbrace{0000}_{0}\ \underbrace{0100}_{4}\ \underbrace{1001}_{9}\ \underbrace{0000}_{0}\ \underbrace{1111}_{\mathrm{f}}\ \underbrace{1101}_{\mathrm{d}}\ \underbrace{1011}_{\mathrm{b}} \quad ,$$

or in "hex" digit form:

$$fl(\pi) \quad = \quad 40490\mathrm{fdb} \qquad .$$

(On some machines this displays only in upper-case letters, i.e. as 40490FDB .)

The storage of other numerical variables, e.g. double precision and quadruple precision, follows a similar pattern to the storage of a single precision real numbers, except the additional length is used to accommodate more significant digits and a larger range of exponents.

The representation of positive integer variables is much cleaner. A positive integer is represented by first converting the integer to a (maximum) thirty-one bit binary number, then appending the sign bit (0) to the front, thus creating a thirty-two bit number, which in turn can be converted into eight four-bit hexadecimal digits. Thus, for example, with the sign bit appended to the front:

$$293 \ = \ (100100101)_2 \ = \ (00000000000000000000000100100101)_2$$

which then segments as:

$$\underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0001}_{1}\ \underbrace{0010}_{2}\ \underbrace{0101}_{5} \quad ,$$

or 00000125 is the hexadecimal IEEE stored form of 293 . (The storage of negative integers is not so clean, and involves the concept of twos complement arithmetic.)